

Efficient Tabular LR Parsing

Mark-Jan Nederhof

Faculty of Arts
University of Groningen
P.O. Box 716
9700 AS Groningen
The Netherlands
markjan@let.rug.nl

Giorgio Satta

Dipartimento di Elettronica ed Informatica
Università di Padova
via Gradenigo, 6/A
I-35131 Padova
Italy
satta@dei.unipd.it

Abstract

We give a new treatment of tabular LR parsing, which is an alternative to Tomita's generalized LR algorithm. The advantage is twofold. Firstly, our treatment is conceptually more attractive because it uses simpler concepts, such as grammar transformations and standard tabulation techniques also known as *chart parsing*. Secondly, the static and dynamic complexity of parsing, both in space and time, is significantly reduced.

1 Introduction

The efficiency of LR(k) parsing techniques (Sippu and Soisalon-Soininen, 1990) is very attractive from the perspective of natural language processing applications. This has stimulated the computational linguistics community to develop extensions of these techniques to general context-free grammar parsing. The best-known example is *generalized* LR parsing, also known as Tomita's algorithm, described by Tomita (1986) and further investigated by, for example, Tomita (1991) and Nederhof (1994a). Despite appearances, the graph-structured stacks used to describe Tomita's algorithm differ very little from *parse tables*, or in other words, generalized LR parsing is one of the so called *tabular* parsing algorithms, among which also the CYK algorithm (Harrison, 1978) and Earley's algorithm (Earley, 1970) can be found. (Tabular parsing is also known as *chart parsing*.)

In this paper we investigate the extension of LR parsing to general context-free grammars from a more general viewpoint: tabular algorithms can often be described by the composition of two constructions. One example is given by Lang (1974) and Billot and Lang (1989): the construction of push-down automata from grammars and the simulation

of these automata by means of tabulation yield different tabular algorithms for different such constructions. Another example, on which our presentation is based, was first suggested by Leermakers (1989): a grammar is first transformed and then a standard tabular algorithm along with some filtering condition is applied using the transformed grammar. In our case, the transformation and the subsequent application of the tabular algorithm result in a new form of tabular LR parsing.

Our method is more efficient than Tomita's algorithm in two respects. First, reduce operations are implemented in an efficient way, by splitting them into several, more primitive, operations (a similar idea has been proposed by Kipps (1991) for Tomita's algorithm). Second, several paths in the computation that must be simulated separately by Tomita's algorithm are collapsed into a single computation path, using state minimization techniques. Experiments on practical grammars have indicated that there is a significant gain in efficiency, with regard to both space and time requirements.

Our grammar transformation produces a so called *cover* for the input grammar, which together with the filtering condition fully captures the specification of the method, abstracting away from algorithmic details such as data structures and control flow. Since this cover can be easily precomputed, implementing our LR parser simply amounts to running the standard tabular algorithm. This is very attractive from an application-oriented perspective, since many actual systems for natural language processing are based on these kinds of parsing algorithm.

The remainder of this paper is organized as follows. In Section 2 some preliminaries are discussed. We review the notion of LR automaton in Section 3 and introduce the notion of 2LR automaton in Section 4. Then we specify our tabular LR method in Section 5, and provide an analysis of the algorithm in Section 6. Finally, some empirical results are given

in Section 7, and further discussion of our method is provided in Section 8.

2 Definitions

Throughout this paper we use standard formal language notation. We assume that the reader is familiar with context-free grammar parsing theory (Harrison, 1978).

A context-free grammar (CFG) is a 4-tuple $G = (\Sigma, N, P, S)$, where Σ and N are two finite disjoint sets of terminal and nonterminal symbols, respectively, $S \in N$ is the start symbol, and P is a finite set of rules. Each rule has the form $A \rightarrow \alpha$ with $A \in N$ and $\alpha \in V^*$, where V denotes $N \cup \Sigma$. The size of G , written $|G|$, is defined as $\sum_{(A \rightarrow \alpha) \in P} |A\alpha|$; by $|\alpha|$ we mean the length of a string of symbols α .

We generally use symbols A, B, C, \dots to range over N , symbols a, b, c, \dots to range over Σ , symbols X, Y, Z to range over V , symbols $\alpha, \beta, \gamma, \dots$ to range over V^* , and symbols v, w, x, \dots to range over Σ^* . We write ϵ to denote the empty string.

A CFG is said to be in *binary form* if $\alpha \in \{\epsilon\} \cup V \cup N^2$ for all of its rules $A \rightarrow \alpha$. (The binary form does not limit the (weak) generative capacity of context-free grammars (Harrison, 1978).) For technical reasons, we sometimes use the *augmented* grammar associated with G , defined as $G^\dagger = (\Sigma^\dagger, N^\dagger, P^\dagger, S^\dagger)$, where $S^\dagger, \triangleright$ and \triangleleft are fresh symbols, $\Sigma^\dagger = \Sigma \cup \{\triangleright, \triangleleft\}$, $N^\dagger = N \cup \{S^\dagger\}$ and $P^\dagger = P \cup \{S^\dagger \rightarrow \triangleright S \triangleleft\}$.

A pushdown automaton (PDA) is a 5-tuple $\mathcal{A} = (\Sigma, Q, T, q_{in}, q_{fin})$, where Σ, Q and T are finite sets of input symbols, stack symbols and transitions, respectively; $q_{in} \in Q$ is the initial stack symbol and $q_{fin} \in Q$ is the final stack symbol.¹ Each transition has the form $\delta_1 \xrightarrow{z} \delta_2$, where $\delta_1, \delta_2 \in Q^*$, $1 \leq |\delta_1|$, $1 \leq |\delta_2| \leq 2$, and $z = \epsilon$ or $z = a$. We generally use symbols q, r, s, \dots to range over Q , and the symbol δ to range over Q^* .

Consider a fixed input string $v \in \Sigma^*$. A *configuration* of the automaton is a pair (δ, w) consisting of a stack $\delta \in Q^*$ and the remaining input w , which is a suffix of the input string v . The rightmost symbol of δ represents the top of the stack. The *initial* configuration has the form (q_{in}, v) , where the stack is formed by the initial stack symbol. The *final* configuration has the form $(q_{in} q_{fin}, \epsilon)$, where the stack is formed by the final stack symbol stacked upon the initial stack symbol.

¹ We dispense with the notion of *state*, traditionally incorporated in the definition of PDA. This does not affect the power of these devices, since states can be encoded within stack symbols and transitions.

The application of a transition $\delta_1 \xrightarrow{z} \delta_2$ is described as follows. If the top-most symbols of the stack are δ_1 , then these symbols may be replaced by δ_2 , provided that either $z = \epsilon$, or $z = a$ and a is the first symbol of the remaining input. Furthermore, if $z = a$ then a is removed from the remaining input. Formally, for a fixed PDA \mathcal{A} we define the binary relation \vdash on configurations as the least relation satisfying $(\delta\delta_1, w) \vdash (\delta\delta_2, w)$ if there is a transition $\delta_1 \xrightarrow{\epsilon} \delta_2$, and $(\delta\delta_1, aw) \vdash (\delta\delta_2, w)$ if there is a transition $\delta_1 \xrightarrow{a} \delta_2$. The recognition of a certain input v is obtained if starting from the initial configuration for that input we can reach the final configuration by repeated application of transitions, or, formally, if $(q_{in}, v) \vdash^* (q_{in} q_{fin}, \epsilon)$, where \vdash^* denotes the reflexive and transitive closure of \vdash .

By a *computation* of a PDA we mean a sequence $(q_{in}, v) \vdash (\delta_1, w_1) \vdash \dots \vdash (\delta_n, w_n)$, $n \geq 0$. A PDA is called *deterministic* if for all possible configurations at most one transition is applicable. A PDA is said to be in *binary form* if, for all transitions $\delta_1 \xrightarrow{z} \delta_2$, we have $|\delta_1| \leq 2$.

3 LR automata

Let $G = (\Sigma, N, P, S)$ be a CFG. We recall the notion of LR automaton, which is a particular kind of PDA. We make use of the augmented grammar $G^\dagger = (\Sigma^\dagger, N^\dagger, P^\dagger, S^\dagger)$ introduced in Section 2.

Let $I_{LR} = \{A \rightarrow \alpha \bullet \beta \mid (A \rightarrow \alpha\beta) \in P^\dagger\}$. We introduce the function *closure* from $2^{I_{LR}}$ to $2^{I_{LR}}$ and the function *goto* from $2^{I_{LR}} \times V$ to $2^{I_{LR}}$. For any $q \subseteq I_{LR}$, *closure*(q) is the smallest set such that

- (i) $q \subseteq \text{closure}(q)$; and
- (ii) $(B \rightarrow \alpha \bullet A\beta) \in \text{closure}(q)$ and $(A \rightarrow \gamma) \in P^\dagger$ together imply $(A \rightarrow \bullet \gamma) \in \text{closure}(q)$.

We then define

$$\begin{aligned} \text{goto}(q, X) = \\ \{A \rightarrow \alpha X \bullet \beta \mid (A \rightarrow \alpha \bullet X\beta) \in \text{closure}(q)\}. \end{aligned}$$

We construct a finite set \mathcal{R}_{LR} as the smallest collection of sets satisfying the conditions:

- (i) $\{S^\dagger \rightarrow \triangleright \bullet S \triangleleft\} \in \mathcal{R}_{LR}$; and
- (ii) for every $q \in \mathcal{R}_{LR}$ and $X \in V$, we have $\text{goto}(q, X) \in \mathcal{R}_{LR}$, provided $\text{goto}(q, X) \neq \emptyset$.

Two elements from \mathcal{R}_{LR} deserve special attention: $q_{in} = \{S^\dagger \rightarrow \triangleright \bullet S \triangleleft\}$, and q_{fin} , which is defined to be the unique set in \mathcal{R}_{LR} containing $(S^\dagger \rightarrow \triangleright S \bullet \triangleleft)$; in other words, $q_{fin} = \text{goto}(q_{in}, S)$.

For $A \in N$, an A -redex is a string $q_0q_1q_2 \cdots q_m$, $m \geq 0$, of elements from \mathcal{R}_{LR} , satisfying the following conditions:

- (i) $(A \rightarrow \alpha \bullet) \in \text{closure}(q_m)$, for some $\alpha = X_1X_2 \cdots X_m$; and
- (ii) $\text{goto}(q_{k-1}, X_k) = q_k$, for $1 \leq k \leq m$.

Note that in such an A -redex, $(A \rightarrow \bullet X_1X_2 \cdots X_m) \in \text{closure}(q_0)$, and $(A \rightarrow X_1 \cdots X_k \bullet X_{k+1} \cdots X_m) \in q_k$, for $0 < k \leq m$.

The LR automaton associated with G is now introduced.

Definition 1 $\mathcal{A}_{LR} = (\Sigma, Q_{LR}, T_{LR}, q_{in}, q_{fin})$, where $Q_{LR} = \mathcal{R}_{LR}$, $q_{in} = \{S^\dagger \rightarrow \triangleright \bullet S \triangleleft\}$, $q_{fin} = \text{goto}(q_{in}, S)$, and T_{LR} contains:

- (i) $q \xrightarrow{a} q'$, for every $a \in \Sigma$ and $q, q' \in \mathcal{R}_{LR}$ such that $q' = \text{goto}(q, a)$;
- (ii) $q\delta \xrightarrow{\epsilon} q'$, for every $A \in N$, A -redex $q\delta$, and $q' \in \mathcal{R}_{LR}$ such that $q' = \text{goto}(q, A)$.

Transitions in (i) above are called *shift*, transitions in (ii) are called *reduce*.

4 2LR Automata

The automata \mathcal{A}_{LR} defined in the previous section are deterministic only for a subset of the CFGs, called the LR(0) grammars (Sippu and Soisalon-Soininen, 1990), and behave nondeterministically in the general case. When designing tabular methods that simulate nondeterministic computations of \mathcal{A}_{LR} , two main difficulties are encountered:

- A reduce transition in \mathcal{A}_{LR} is an elementary operation that removes from the stack a number of elements bounded by the size of the underlying grammar. Consequently, the time requirement of tabular simulation of \mathcal{A}_{LR} computations can be onerous, for reasons pointed out by Sheil (1976) and Kipps (1991).
- The set \mathcal{R}_{LR} can be exponential in the size of the grammar (Johnson, 1991). If in such a case the computations of \mathcal{A}_{LR} touch upon each state, then time and space requirements of tabular simulation are obviously onerous.

The first issue above is solved here by recasting \mathcal{A}_{LR} in binary form. This is done by considering each reduce transition as a sequence of “pop” operations which affect at most two stack symbols at a time. (See also Lang (1974), Villemonte de la Clergerie (1993) and

Nederhof (1994a), and for LR parsing specifically Kipps (1991) and Leermakers (1992b).) The following definition introduces this new kind of automaton.

Definition 2 $\mathcal{A}'_{LR} = (\Sigma, Q'_{LR}, T'_{LR}, q_{in}, q_{fin})$, where $Q'_{LR} = \mathcal{R}_{LR} \cup I_{LR}$, $q_{in} = \{S^\dagger \rightarrow \triangleright \bullet S \triangleleft\}$, $q_{fin} = \text{goto}(q_{in}, S)$ and T'_{LR} contains:

- (i) $q \xrightarrow{a} q'$, for every $a \in \Sigma$ and $q, q' \in \mathcal{R}_{LR}$ such that $q' = \text{goto}(q, a)$;
- (ii) $q \xrightarrow{\epsilon} q' (A \rightarrow \alpha \bullet)$, for every $q \in \mathcal{R}_{LR}$ and $(A \rightarrow \alpha \bullet) \in \text{closure}(q)$;
- (iii) $q (A \rightarrow \alpha X \bullet \beta) \xrightarrow{\epsilon} (A \rightarrow \alpha \bullet X \beta)$, for every $q \in \mathcal{R}_{LR}$ and $(A \rightarrow \alpha X \bullet \beta) \in q$;
- (iv) $q (A \rightarrow \bullet \alpha) \xrightarrow{\epsilon} q'$, for every $q, q' \in \mathcal{R}_{LR}$ and $(A \rightarrow \alpha) \in P^\dagger$ such that $q' = \text{goto}(q, A)$.

Transitions in (i) above are again called *shift*, transitions in (ii) are called *initiate*, those in (iii) are called *gathering*, and transitions in (iv) are called *goto*. The role of a reduce step in \mathcal{A}_{LR} is taken over in \mathcal{A}'_{LR} by an initiate step, a number of gathering steps, and a goto step. Observe that these steps involve the new stack symbols $(A \rightarrow \alpha \bullet \beta) \in I_{LR}$ that are distinguishable from possible stack symbols $\{A \rightarrow \alpha \bullet \beta\} \in \mathcal{R}_{LR}$.

We now turn to the second above-mentioned problem, regarding the size of set \mathcal{R}_{LR} . The problem is in part solved here as follows. The number of states in \mathcal{R}_{LR} is considerably reduced by identifying two states if they become identical after items $A \rightarrow \alpha \bullet \beta$ from I_{LR} have been simplified to only the suffix of the right-hand side β . This is reminiscent of techniques of state minimization for finite automata (Booth, 1967), as they have been applied before to LR parsing, e.g., by Pager (1970) and Nederhof and Sarbo (1993).

Let G^\dagger be the augmented grammar associated with a CFG G , and let $I_{2LR} = \{\beta \mid (A \rightarrow \alpha \beta) \in P^\dagger\}$. We define variants of the *closure* and *goto* functions from the previous section as follows. For any set $q \subseteq I_{2LR}$, $\text{closure}'(q)$ is the smallest collection of sets such that

- (i) $q \subseteq \text{closure}'(q)$; and
- (ii) $(A\beta) \in \text{closure}'(q)$ and $(A \rightarrow \gamma) \in P^\dagger$ together imply $(\gamma) \in \text{closure}'(q)$.

Also, we define

$$\text{goto}'(q, X) = \{\beta \mid (X\beta) \in \text{closure}'(q)\}.$$

We now construct a finite set \mathcal{R}_{2LR} as the smallest set satisfying the conditions:

- (i) $\{S\triangleleft\} \in \mathcal{R}_{2LR}$; and
- (ii) for every $q \in \mathcal{R}_{2LR}$ and $X \in V$, we have $goto'(q, X) \in \mathcal{R}_{2LR}$, provided $goto'(q, X) \neq \emptyset$.

As stack symbols, we take the elements from I_{2LR} and a subset of elements from $(V \times \mathcal{R}_{2LR})$:

$$Q_{2LR} = \{(X, q) \mid \exists q'[goto'(q', X) = q]\} \cup I_{2LR}$$

In a stack symbol of the form (X, q) , the X serves to record the grammar symbol that has been recognized last, cf. the symbols that formerly were found immediately before the dots.

The 2LR automaton associated with G can now be introduced.

Definition 3 $\mathcal{A}_{2LR} = (\Sigma, Q_{2LR}, T_{2LR}, q'_{in}, q'_{fin})$, where Q_{2LR} is as defined above, $q'_{in} = (\triangleright, \{S\triangleleft\})$, $q'_{fin} = (S, goto'(\{S\triangleleft\}, S))$, and T_{2LR} contains:

- (i) $(X, q) \xrightarrow{a} (X, q) (a, q')$, for every $a \in \Sigma$ and $(X, q), (a, q') \in Q_{2LR}$ such that $q' = goto'(q, a)$;
- (ii) $(X, q) \xrightarrow{\epsilon} (X, q) (\epsilon)$, for every $(X, q) \in Q_{2LR}$ such that $\epsilon \in closure'(q)$;
- (iii) $(X, q) (\beta) \xrightarrow{\epsilon} (X\beta)$, for every $(X, q) \in Q_{2LR}$ and $\beta \in q$;
- (iv) $(X, q) (\alpha) \xrightarrow{\epsilon} (X, q) (A, q')$, for every $(X, q), (A, q') \in Q_{2LR}$ and $(A \rightarrow \alpha) \in P^\dagger$ such that $q' = goto'(q, A)$.

Note that in the case of a reduce/reduce conflict with two grammar rules sharing some suffix in the right-hand side, the gathering steps of \mathcal{A}_{2LR} will treat both rules simultaneously, until the parts of the right-hand sides are reached where the two rules differ. (See Leermakers (1992a) for a similar sharing of computation for common suffixes.)

An interesting fact is that the automaton \mathcal{A}_{2LR} is very similar to the automaton \mathcal{A}_{LR} constructed for a grammar transformed by the transformation τ_{two} given by Nederhof and Satta (1994).²

5 The algorithm

This section presents a tabular LR parser, which is the main result of this paper. The parser is derived from the 2LR automata introduced in the previous section. Following the general approach presented by Leermakers (1989), we simulate computations of

²For the earliest mention of this transformation, we have encountered pointers to Schauerte (1973). Regrettably, we have as yet not been able to get hold of a copy of this paper.

these devices using a tabular method, a grammar transformation and a filtering function.

We make use of a tabular parsing algorithm which is basically an asynchronous version of the CYK algorithm, as presented by Harrison (1978), extended to productions of the forms $A \rightarrow B$ and $A \rightarrow \epsilon$ and with a left-to-right filtering condition. The algorithm uses a parse table consisting in a 0-indexed square array U . The indices represent positions in the input string. We define U_i to be $\bigcup_{k \leq i} U_{k,i}$.

Computation of the entries of U is moderated by a filtering process. This process makes use of a function $pred$ from 2^N to 2^N , specific to a certain context-free grammar. We have a certain nonterminal A_{init} which is initially inserted in $U_{0,0}$ in order to start the recognition process.

We are now ready to give a formal specification of the tabular algorithm.

Algorithm 1 Let $G = (\Sigma, N, P, S)$ be a CFG in binary form, let $pred$ be a function from 2^N to 2^N , let A_{init} be the distinguished element from N , and let $v = a_1 a_2 \dots a_n \in \Sigma^*$ be an input string. We compute the least $(n+1) \times (n+1)$ table U such that $A_{init} \in U_{0,0}$ and

- (i) $A \in U_{j-1,j}$
if $(A \rightarrow a_j) \in P, A \in pred(U_{j-1})$;
- (ii) $A \in U_{j,j}$
if $(A \rightarrow \epsilon) \in P, A \in pred(U_j)$;
- (iii) $A \in U_{i,j}$
if $B \in U_{i,k}, C \in U_{k,j}, (A \rightarrow BC) \in P, A \in pred(U_i)$;
- (iv) $A \in U_{i,j}$
if $B \in U_{i,j}, (A \rightarrow B) \in P, A \in pred(U_i)$.

The string has been accepted when $S \in U_{0,n}$.

We now specify a grammar transformation, based on the definition of \mathcal{A}_{2LR} .

Definition 4 Let $\mathcal{A}_{2LR} = (\Sigma, Q_{2LR}, T_{2LR}, q'_{in}, q'_{fin})$ be the 2LR automaton associated with a CFG G . The **2LR cover** associated with G is the CFG $\mathcal{C}_{2LR}(G) = (\Sigma, Q_{2LR}, P_{2LR}, q'_{fin})$, where the rules in P_{2LR} are given by:

- (i) $(a, q') \rightarrow a$,
for every $(X, q) \xrightarrow{a} (X, q) (a, q') \in T_{2LR}$;
- (ii) $(\epsilon) \rightarrow \epsilon$,
for every $(X, q) \xrightarrow{\epsilon} (X, q) (\epsilon) \in T_{2LR}$;
- (iii) $(X\beta) \rightarrow (X, q) (\beta)$,
for every $(X, q) (\beta) \xrightarrow{\epsilon} (X\beta) \in T_{2LR}$;

- (iv) $(A, q') \rightarrow (\alpha)$,
for every $(X, q) (\alpha) \xrightarrow{\epsilon} (X, q) (A, q') \in T_{2LR}$.

Observe that there is a direct, one-to-one correspondence between transitions of \mathcal{A}_{2LR} and productions of $\mathcal{C}_{2LR}(G)$.

The accompanying function $pred$ is defined as follows (q, q', q'' range over the stack elements):

$$\begin{aligned} pred(\tau) = & \{q \mid q'q'' \xrightarrow{\epsilon} q \in T_{2LR}\} \cup \\ & \{q \mid q' \in \tau, q' \xrightarrow{z} q' q \in T_{2LR}\} \cup \\ & \{q \mid q' \in \tau, q' q'' \xrightarrow{\epsilon} q' q \in T_{2LR}\}. \end{aligned}$$

The above definition implies that only the tabular equivalents of the shift, initiate and goto transitions are subject to actual filtering; the simulation of the gathering transitions does not depend on elements in τ .

Finally, the distinguished nonterminal from the cover used to initialize the table is q'_{in} . Thus we start with $(\triangleright, \{S\triangleleft\}) \in U_{0,0}$.

The 2LR cover introduces spurious ambiguity: where some grammar G would allow a certain number of parses to be found for a certain input, the grammar $\mathcal{C}_{2LR}(G)$ in general allows *more* parses. This problem is in part solved by the filtering function $pred$. The remaining spurious ambiguity is avoided by a particular way of constructing the parse trees, described in what follows.

After Algorithm 1 has recognized a given input, the set of all parse trees can be computed as $tree(q'_{fin}, 0, n)$ where the function $tree$, which determines sets of either parse trees or lists of parse trees for entries in U , is recursively defined by:

- (i) $tree((a, q'), i, j)$ is the set $\{a\}$. This set contains a single parse tree consisting of a single node labelled a .
- (ii) $tree(\epsilon, i, i)$ is the set $\{\epsilon\}$. This set consists of an empty list of trees.
- (iii) $tree(X\beta, i, j)$ is the union of the sets $\mathcal{T}_{(X\beta), i, j}^k$, where $i \leq k \leq j$, $(\beta) \in U_{k, j}$, and there is at least one $(X, q) \in U_{i, k}$ and $(X\beta) \rightarrow (X, q) (\beta)$ in $\mathcal{C}_{2LR}(G)$, for some q . For each such k , select one such q . We define $\mathcal{T}_{(X\beta), i, j}^k = \{t \cdot ts \mid t \in tree((X, q), i, k) \wedge ts \in tree(\beta, k, j)\}$. Each $t \cdot ts$ is a list of trees, with head t and tail ts .
- (iv) $tree((A, q'), i, j)$ is the union of the sets $\mathcal{T}_{(A, q'), i, j}^\alpha$, where $(\alpha) \in U_{i, j}$ is such that $(A, q') \rightarrow (\alpha)$ in $\mathcal{C}_{2LR}(G)$. We define $\mathcal{T}_{(A, q'), i, j}^\alpha = \{glue(A, ts) \mid ts \in tree(\alpha, i, j)\}$. The function $glue$ constructs a tree from a fresh root node

labelled A and the trees in list ts as immediate subtrees.

We emphasize that in the third clause above, one should not consider more than one q for given k in order to prevent spurious ambiguity. (In fact, for fixed X, i, k and for different q such that $(X, q) \in U_{i, k}$, $tree((X, q), i, k)$ yields the exact same set of trees.) With this proviso, the degree of ambiguity, i.e. the number of parses found by the algorithm for any input, is reduced to exactly that of the source grammar.

A practical implementation would construct the parse trees on-the-fly, attaching them to the table entries, allowing packing and sharing of subtrees (cf. the literature on *parse forests* (Tomita, 1986; Billo and Lang, 1989)). Our algorithm actually only needs one (packed) subtree for several $(X, q) \in U_{i, k}$ with fixed X, i, k but different q . The resulting parse forests would then be optimally compact, contrary to some other LR-based tabular algorithms, as pointed out by Rekers (1992), Nederhof (1993) and Nederhof (1994b).

6 Analysis of the algorithm

In this section, we investigate how the steps performed by Algorithm 1 (applied to the 2LR cover) relate to those performed by \mathcal{A}_{2LR} , for the same input.

We define a subrelation \models^+ of \vdash^+ as: $(\delta, uw) \models^+ (\delta\delta', w)$ if and only if $(\delta, uw) = (\delta, z_1 z_2 \dots z_m w) \vdash (\delta\delta_1, z_2 \dots z_m w) \vdash \dots \vdash (\delta\delta_m, w) = (\delta\delta', w)$, for some $m \geq 1$, where $|\delta_k| > 0$ for all k , $1 \leq k \leq m$. Informally, we have $(\delta, uw) \models^+ (\delta\delta', w)$ if configuration $(\delta\delta', w)$ can be reached from (δ, uw) without the bottom-most part δ of the intermediate stacks being affected by any of the transitions; furthermore, at least one element is pushed on top of δ .

The following characterization relates the automaton \mathcal{A}_{2LR} and Algorithm 1 applied to the 2LR cover. Symbol $q \in Q_{2LR}$ is eventually added to $U_{i, j}$ if and only if for some δ :

$$(q'_{in}, a_1 \dots a_n) \vdash^* (\delta, a_{i+1} \dots a_n) \models^+ (\delta q, a_{j+1} \dots a_n).$$

In words, q is found in entry $U_{i, j}$ if and only if, at input position j , the automaton would push some element q on top of some lower-part of the stack δ that remains unaffected while the input from i to j is being read.

The above characterization, whose proof is not reported here, is the justification for calling the resulting algorithm tabular LR parsing. In particular, for a grammar for which \mathcal{A}_{2LR} is deterministic, i.e. for an LR(0) grammar, the number of steps performed

by \mathcal{A}_{2LR} and the number of steps performed by the above algorithm are exactly the same. In the case of grammars which are not LR(0), the tabular LR algorithm is more efficient than for example a backtrack realisation of \mathcal{A}_{2LR} .

For determining the order of the time complexity of our algorithm, we look at the most expensive step, which is the computation of an element $(X\beta) \in U_{i,j}$ from two elements $(X, q) \in U_{i,k}$ and $(\beta) \in U_{k,j}$, through $(X, q) (\beta) \xrightarrow{\epsilon} (X\beta) \in T_{2LR}$. In a straightforward realisation of the algorithm, this step can be applied $\mathcal{O}(|T_{2LR}| \cdot |v|^3)$ times (once for each i, k, j and each transition), each step taking a constant amount of time. We conclude that the time complexity of our algorithm is $\mathcal{O}(|T_{2LR}| \cdot |v|^3)$.

As far as space requirements are concerned, each set $U_{i,j}$ or U_i contains at most $|Q_{2LR}|$ elements. (One may assume an auxiliary table storing each U_i .) This results in a space complexity $\mathcal{O}(|Q_{2LR}| \cdot |v|^2)$.

The entries in the table represent single stack elements, as opposed to pairs of stack elements following Lang (1974) and Leermakers (1989). This has been investigated before by Nederhof (1994a, p. 25) and Villemonte de la Clergerie (1993, p. 155).

7 Empirical results

We have performed some experiments with Algorithm 1 applied to \mathcal{A}_{2LR} and \mathcal{A}'_{LR} , for 4 practical context-free grammars. For \mathcal{A}'_{LR} a cover was used analogous to the one in Definition 4; the filtering function remains the same.

The first grammar generates a subset of the programming language ALGOL 68 (van Wijngaarden and others, 1975). The second and third grammars generate a fragment of Dutch, and are referred to as the CORRie grammar (Vosse, 1994) and the Deltra grammar (Schoorl and Belder, 1990), respectively. These grammars were stripped of their arguments in order to convert them into context-free grammars. The fourth grammar, referred to as the Alvey grammar (Carroll, 1993), generates a fragment of English and was automatically generated from a unification-based grammar.

The test sentences have been obtained by automatic generation from the grammars, using the Grammar Workbench (Nederhof and Koster, 1992), which uses a random generator to select rules; therefore these sentences do not necessarily represent input typical of the applications for which the grammars were written. Table 1 summarizes the test material.

Our implementation is merely a prototype, which means that absolute duration of the parsing process

$G = (\Sigma, N, P, S)$	$ G $	$ N $	$ P $	$ w $
ALGOL 68	783	167	330	13.7
CORRie	1141	203	424	12.3
Deltra	1929	281	703	10.8
Alvey	5072	265	1484	10.7

Table 1: The test material: the four grammars and some of their dimensions, and the average length of the test sentences (20 sentences of various length for each grammar).

G	\mathcal{A}'_{LR}		\mathcal{A}_{2LR}	
	space	time	space	time
ALGOL 68	327	375	234	343
CORRie	7548	28028	5131	22414
Deltra	11772	94824	6526	70333
Alvey	599	1147	354	747

Table 2: Dynamic requirements: average space and time per sentence.

is little indicative of the actual efficiency of more sophisticated implementations. Therefore, our measurements have been restricted to implementation-independent quantities, viz. the number of elements stored in the parse table and the number of elementary steps performed by the algorithm. In a practical implementation, such quantities will strongly influence the space and time complexity, although they do not represent the only determining factors. Furthermore, all optimizations of the time and space efficiency have been left out of consideration.

Table 2 presents the costs of parsing the test sentences. The first and third columns give the number of entries stored in table U , the second and fourth columns give the number of elementary steps that were performed.

An elementary step consists of the derivation of one element in Q'_{LR} or Q_{2LR} from one or two other elements. The elements that are used in the filtering process are counted individually. We give an example for the case of \mathcal{A}'_{LR} . Suppose we derive an element $q' \in U_{i,j}$ from an element $(A \rightarrow \bullet \alpha) \in U_{i,j}$, warranted by two elements $q_1, q_2 \in U_i$, $q_1 \neq q_2$, through *pred*, in the presence of q_1 $(A \rightarrow \bullet \alpha) \xrightarrow{\epsilon} q_1 q' \in T'_{LR}$ and $q_2 (A \rightarrow \bullet \alpha) \xrightarrow{\epsilon} q_2 q' \in T'_{LR}$. We then count *two* parsing steps, one for q_1 and one for q_2 .

Table 2 shows that there is a significant gain in space and time efficiency when moving from \mathcal{A}'_{LR} to

G	\mathcal{A}'_{LR}			$\mathcal{A}_{2\text{LR}}$		
	$ \mathcal{R}_{\text{LR}} $	$ \mathcal{Q}'_{\text{LR}} $	$ T'_{\text{LR}} $	$ \mathcal{R}_{2\text{LR}} $	$ \mathcal{Q}_{2\text{LR}} $	$ T_{2\text{LR}} $
ALGOL 68	434	1,217	13,844	109	724	12,387
CORRie	600	1,741	22,129	185	821	15,569
Deltra	856	2,785	54,932	260	1,089	37,510
Alvey	3,712	8,784	1,862,492	753	3,065	537,852

Table 3: Static requirements.

$\mathcal{A}_{2\text{LR}}$.

Apart from the dynamic costs of parsing, we have also measured some quantities relevant to the construction and storage of the two types of tabular LR parser. These data are given in Table 3.

We see that the number of states is strongly reduced with regard to traditional LR parsing. In the case of the Alvey grammar, moving from $|\mathcal{R}_{\text{LR}}|$ to $|\mathcal{R}_{2\text{LR}}|$ amounts to a reduction to 20.3 %. Whereas time- and space-efficient computation of \mathcal{R}_{LR} for this grammar is a serious problem, computation of $\mathcal{R}_{2\text{LR}}$ will not be difficult on any modern computer. Also significant is the reduction from $|T'_{\text{LR}}|$ to $|T_{2\text{LR}}|$, especially for the larger grammars. These quantities correlate with the amount of storage needed for naive representation of the respective automata.

8 Discussion

Our treatment of tabular LR parsing has two important advantages over the one by Tomita:

- It is conceptually simpler, because we make use of simple concepts such as a grammar transformation and the well-understood CYK algorithm, instead of a complicated mechanism working on graph-structured stacks.
- Our algorithm requires fewer LR states. This leads to faster parser generation, to smaller parsers, and to reduced time and space complexity of parsing itself.

The conceptual simplicity of our formulation of tabular LR parsing allows comparison with other tabular parsing techniques, such as Earley’s algorithm (Earley, 1970) and tabular left-corner parsing (Nederhof, 1993), based on implementation-independent criteria. This is in contrast to experiments reported before (e.g. by Shann (1991)), which treated tabular LR parsing differently from the other techniques.

The reduced time and space complexities reported in the previous section pertain to the tabular realisation of two parsing techniques, expressed by the

automata \mathcal{A}'_{LR} and $\mathcal{A}_{2\text{LR}}$. The tabular realisation of the former automata is very close to a variant of Tomita’s algorithm by Kipps (1991). The objective of our experiments was to show that the automata $\mathcal{A}_{2\text{LR}}$ provide a better basis than \mathcal{A}'_{LR} for tabular LR parsing with regard to space and time complexity.

Parsing algorithms that are not based on the LR technique have however been left out of consideration, and so were techniques for unification grammars and techniques incorporating finite-state processes.³

Theoretical considerations (Leermakers, 1989; Schabes, 1991; Nederhof, 1994b) have suggested that for natural language parsing, LR-based techniques may not necessarily be superior to other parsing techniques, although convincing empirical data to this effect has never been shown. This issue is difficult to resolve because so much of the relative efficiency of the different parsing techniques depends on particular grammars and particular input, as well as on particular implementations of the techniques. We hope the conceptual framework presented in this paper may at least partly alleviate this problem.

Acknowledgements

The first author is supported by the Dutch Organization for Scientific Research (NWO), under grant 305-00-802. Part of the present research was done while the second author was visiting the Center for Language and Speech Processing, Johns Hopkins University, Baltimore, MD.

We received kind help from John Carroll, Job Honig, Kees Koster, Theo Vosse and Hans de Vreught in finding the grammars mentioned in this paper. Generous help with locating relevant literature was provided by Anton Nijholt, Rockford Ross, and Arnd Rußmann.

³As remarked before by Nederhof (1993), the algorithms by Schabes (1991) and Leermakers (1989) are not really related to LR parsing, although some notation used in these papers suggests otherwise.

References

- [Billot and Lang1989] Billot, S. and B. Lang. 1989. The structure of shared forests in ambiguous parsing. In *27th Annual Meeting of the ACL*, pages 143–151.
- [Booth1967] Booth, T.L. 1967. *Sequential Machines and Automata Theory*. Wiley, New York.
- [Carroll1993] Carroll, J.A. 1993. Practical unification-based parsing of natural language. Technical Report No. 314, University of Cambridge, Computer Laboratory, England. PhD thesis.
- [Earley1970] Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.
- [Harrison1978] Harrison, M.A. 1978. *Introduction to Formal Language Theory*. Addison-Wesley.
- [Johnson1991] Johnson, M. 1991. The computational complexity of GLR parsing. In Tomita (1991), chapter 3, pages 35–42.
- [Kipps1991] Kipps, J.R. 1991. GLR parsing in time $O(n^3)$. In Tomita (1991), chapter 4, pages 43–59.
- [Lang1974] Lang, B. 1974. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages and Programming, 2nd Colloquium*, LNCS 14, pages 255–269, Saarbrücken. Springer-Verlag.
- [Leermakers1989] Leermakers, R. 1989. How to cover a grammar. In *27th Annual Meeting of the ACL*, pages 135–142.
- [Leermakers1992a] Leermakers, R. 1992a. A recursive ascent Earley parser. *Information Processing Letters*, 41(2):87–91.
- [Leermakers1992b] Leermakers, R. 1992b. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science*, 104:299–312.
- [Nederhof1993] Nederhof, M.J. 1993. Generalized left-corner parsing. In *Sixth Conference of the European Chapter of the ACL*, pages 305–314.
- [Nederhof1994a] Nederhof, M.J. 1994a. *Linguistic Parsing and Program Transformations*. Ph.D. thesis, University of Nijmegen.
- [Nederhof1994b] Nederhof, M.J. 1994b. An optimal tabular parsing algorithm. In *32nd Annual Meeting of the ACL*, pages 117–124.
- [Nederhof and Koster1992] Nederhof, M.J. and K. Koster. 1992. A customized grammar workbench. In J. Aarts, P. de Haan, and N. Oostdijk, editors, *English Language Corpora: Design, Analysis and Exploitation*, Papers from the thirteenth International Conference on English Language Research on Computerized Corpora, pages 163–179, Nijmegen. Rodopi.
- [Nederhof and Sarbo1993] Nederhof, M.J. and J.J. Sarbo. 1993. Increasing the applicability of LR parsing. In *Third International Workshop on Parsing Technologies*, pages 187–201.
- [Nederhof and Satta1994] Nederhof, M.J. and G. Satta. 1994. An extended theory of head-driven parsing. In *32nd Annual Meeting of the ACL*, pages 210–217.
- [Pager1970] Pager, D. 1970. A solution to an open problem by Knuth. *Information and Control*, 17:462–473.
- [Rekers1992] Rekers, J. 1992. *Parser Generation for Interactive Environments*. Ph.D. thesis, University of Amsterdam.
- [Schabes1991] Schabes, Y. 1991. Polynomial time and space shift-reduce parsing of arbitrary context-free grammars. In *29th Annual Meeting of the ACL*, pages 106–113.
- [Schauerte1973] Schauerte, R. 1973. Transformationen von LR(k)-grammatiken. Diplomarbeit, Universität Göttingen, Abteilung Informatik.
- [Schoorl and Belder1990] Schoorl, J.J. and S. Belder. 1990. Computational linguistics at Delft: A status report. Report WTM/TT 90–09, Delft University of Technology, Applied Linguistics Unit.
- [Shann1991] Shann, P. 1991. Experiments with GLR and chart parsing. In Tomita (1991), chapter 2, pages 17–34.
- [Sheil1976] Sheil, B.A. 1976. Observations on context-free parsing. *Statistical Methods in Linguistics*, pages 71–109.
- [Sippu and Soisalon-Soininen1990] Sippu, S. and E. Soisalon-Soininen. 1990. *Parsing Theory, Vol. II: LR(k) and LL(k) Parsing*. Springer-Verlag.
- [Tomita1986] Tomita, M. 1986. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers.
- [Tomita1991] Tomita, M., editor. 1991. *Generalized LR Parsing*. Kluwer Academic Publishers.
- [van Wijngaarden and others1975] van Wijngaarden, A. et al. 1975. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5:1–236.

- [Villemonthe de la Clergerie1993] Villemonthe de la Clergerie, E. 1993. *Automates à Piles et Programmation Dynamique — DyALog: Une application à la Programmation en Logique*. Ph.D. thesis, Université Paris VII.
- [Vosse1994] Vosse, T.G. 1994. *The Word Connection*. Ph.D. thesis, University of Leiden.